Formal Access Control Policy Language (FACPL)

– User's Guide –

Andrea Margheri[1,2], Massimiliano Masi[3], Rosario Pugliese[1], and Francesco Tiezzi[4]

[1] Università degli Studi di Firenze, Viale Morgagni, 65 - 50134 Firenze, Italy
{andrea.margheri,rosario.pugliese}@unifi.it

[2] Università di Pisa, Largo Bruno Pontecorvo, 3 - 56127 Pisa, Italy
margheri@di.unipi.it

[3] Tiani "Spirit" GmbH, Guglgasse, 6 - 1110 Vienna, Austria
massimiliano.masi@tiani-spirit.com

[4] IMT Advanced Studies Lucca, Piazza S. Ponziano, 6 - 55100, Lucca, Italy
francesco.tiezzi@imtlucca.it

Abstract. Formal Access Control Policy Language (FACPL) is an easy-to-learn, tiny language, with a mathematically defined semantics, for writing access control policies and requests. In this guide, we describe the FACPL approach for managing access control and systems' behaviour in policy-based software architectures. Specifically, first we gently introduce FACPL's syntax and semantics in a step-by-step fashion while commenting upon some policies in force in an eHealth scenario, which is gradually introduced throughout the presentation. Then, we present the software tools implemented for developing and enforcing FACPL policies: a powerful Eclipse-based development environment and a Java library supporting the policy evaluation process.

http://rap.dsi.unifi.it/facpl/

Table of Contents

1 The FACPL Language 3
1.1 Syntax 3
1.2 Evaluation Process 7
2 The FACPL library 10
2.1 Library Design 10
2.2 An Example of Java Policy 13
3 The FACPL IDE 14
3.1 The Xtext Framework 15
3.2 Plugin Installation 15
3.3 Creation of a FACPL Project and a FACPL File 16
3.4 FACPL commands and facets 17
4 The eHealth Scenario 21
A The grammar of FACPL 23
B Combining algorithms 24

1 The FACPL Language

FACPL is a policy language that can express access control policies as well as policies dealing with other systems' aspects as, e.g., resource usage and adaptation. It takes inspiration from the OASIS standard eXtensible Access Control Markup Language (XACML) [2], but is simpler and more human readable. In fact, FACPL has a compact and intuitive syntax and is endowed with a formal semantics based on solid mathematical foundations, which make it easy to learn and use the language.

The FACPL language permits defining both *Policy Authorization Frameworks* (PAFs), which specify access control strategies for calculating and enforcing authorisation decisions, and *requests*, which report credentials required to perform actions over resources. A request submitted to an access control system is evaluated according to the policies defined in the PAF. In this way, it is obtained an authorisation decision that determines whether the requestor is allowed or not to perform the action requested on the resource. This decision is then enforced by performing the additional actions (called *obligations*) returned together with the decision. In fact, a PAF defines strategies for calculating authorisation decisions, i.e. a Policy Decision Point (PDP), and for enforcing them, i.e. a Policy Enforcement Point (PEP).

In the rest of the section, we informally present the syntax of the language and the semantics of the decision and enforcement processes, by exploiting as a running example a case study concerning the management of patients' medical records by healthcare professionals, i.e. medical doctors, nurses and administrative people. The formal definition of the language syntax, i.e. grammars of policies and requests, is relegated to Appendix A. We refer the interested reader to [1] for the formal presentation of FACPL's semantics.

1.1 Syntax

A FACPL request is issued by a client to request a service, i.e. an action over a resource. The request, in addition to its identifier, provides the authentication credentials and the specification of the type of the requested service. This information is organised in terms of *attributes*, namely name-value pairs of the form:

(`Identifier/Identifier` , `Literal`)

Names are structured as `Identifier/Identifier`, where the first identifier stands for a category name (as, e.g., `subject`, `resource`, `action`, and `environment`) and the second for an attribute name (as, e.g., `subject-id` and `action-id`). A sample request made by a `medical doctor` for reading a resource with code `34133-9` (i.e., a patient summary) for `TREATMENT` purposes, is written as follows.

```
Request:{Request1
    (subject/purposeofuse, "TREATEMENT")
    (subject/role, "medical doctor")
    (subject/doctor-id, "jh1234")
    (subject/hl7.permission, "PRD-003", "PRD-005", "PRD-010", "PRD-016")
    (resource/resource-id, "34133-9")
    (resource/resource-id.email, "mail@gmail.com")
    (action/action-id, "Read")
}
```

The request is received by the PAF, which calls the PDP for starting the evaluation procedure on the basis of the policies stored. The PDP performs the evaluation of all stored policies and returns one of the following decisions:

– `permit`: the requestor is granted permission to perform the action requested;

– `deny`: the requestor is not granted permission to perform the action;
– `not-applicable`: the PDP does not have any policy that applies to the request, so it cannot calculate an authorisation decision;
– `indeterminate`: the PDP is not able to evaluate the access request (reasons for such inability include, e.g., missing attributes, network errors, policy evaluation errors).

The first two decisions can be paired with a set of *obligations*, i.e., actions that must be successfully performed for enforcing the decision, such as data-processing or notification actions.

The `indeterminate` value is also specialised in order to indicate which could be the resulting decision if errors would not have taken place. The extended indeterminate decisions are:

– `indetP`: the potential decision could be `permit` but not `deny`;
– `indetD`: the potential decision could be `deny` but not `permit`;
– `indetDP`: the potential decision could be `permit` or `deny`.

When the PDP completes the decision process, the resulting authorisation decision, which can include obligations, is sent to the PEP for the enforcement process. To enforce the decision, the PEP must understand and discharge all the included obligations. In case some obligations cannot be discharged, the final decision depends on the *enforcement algorithm* chosen for the PEP. FACPL provides the following algorithms:

– `base`: it allows (resp. forbids) access only if the decision is `permit` (resp. `deny`) and all obligations are successfully discharged, otherwise it enforces `indeterminate`;
– `deny-biased`: if the decision is `permit` and all obligations are successfully discharged, the access is granted, otherwise it is forbidden;
– `permit-biased`: if the decision is `deny` and all obligations are successfully discharged, the access is forbidden, otherwise it is granted.

While the PEP is defined simply by the enforcement algorithm, the PDP is a collection of *policies* and *policy sets* combined together by a *combining algorithm*, which specifies the decision to be returned in case multiple policies apply and return different decisions. For example, two policies are at the basis of the eHealth scenario taken as example for this guide: one forbids all the requestor's attempts and the other one grants only medical doctors to read patient summaries. These policies are grouped in a policy set under the combining algorithm `permit-overrides` (explained below). Therefore, the PAF containing the PDP and the PEP, where the latter relies on the `base` enforcement algorithm, is defined as follows.

```
{
 pep: base
 pdp: permit-overrides
      PolicySet policySet { permit-overrides
         policies:
            Policy denyAll< permit-overrides
               rules: Rule denyAll(deny)
            >
            include PatientSummary
         obl:
            [permit M redirectTo(epsos/redirectto:url)]
      }
}
```

Notably, in the PDP, the policy for the patient summary is included through a cross name reference, i.e. `include PatientSummary`, which simplifies organisation of code. Besides the two policies, the policy set also specifies an obligation defining, in case of `permit` decision, an additional action to be performed in the enforcement process. This element we will be further explained shortly.

The policy combing algorithm specified by a policy set can be chosen among the following ones:

- `deny-overrides`: if any policy in the considered set evaluates to `deny`, then the result of the policy set combination is `deny`. In other words, `deny` takes precedence, regardless of the result of evaluating any other policy in the policy set. Instead, if at least a policy evaluates to `permit` and all the other policies evaluate to `not-applicable`, `permit` or `indetP`, then the result of the combination is `permit`. If all policies are found to be `not-applicable` to the decision request, then the policy set evaluates to `not-applicable`. Finally, a policy evaluation results in `indetDP`, `indetD` and `indetP` in the remaining cases, according to specific error situations.
- `permit-overrides`: this combining algorithm is the dual of the previous one, i.e. this time `permit` takes precedence over the other results.
- `deny-unless-permit`: this algorithm is similar to `permit-overrides`, because it is intended for those cases where a `permit` decision takes precedence over `deny` decisions; differently from `permit-overrides`, this algorithm returns neither `not-applicable` nor an (extended) indeterminate value.
- `permit-unless-deny`: this algorithm is the dual of the previous one, i.e. `deny` takes precedence over `permit` decisions.
- `first-applicable`: in this case, the policies are evaluated in the order of appearance in the policy set and the combined result is the same as the result of evaluating the first policy in the list of policies whose target is applicable to the decision request, if such result is either `permit` or `deny`. If all policies evaluate to `not-applicable`, then the policy set evaluates to `not-applicable`. If an error occurs while evaluating the target of a policy, or a policy evaluation results in `indetDP`, then the evaluation of the algorithm halts and the policy set evaluates to `indetDP`.
- `only-one-applicable`: this algorithm ensures that one and only one policy is applicable by virtue of its target. If no policy applies, then the result is `not-applicable`, but if more than one policy is applicable, then the result is `indetDP`. When exactly one policy is applicable, the result of the combining algorithm is the result of evaluating the single applicable policy. If an error occurs while evaluating the target of a policy, or the policy evaluation results in `indetDP`, then the policy set evaluates to `indetDP`.

These algorithms, except the `only-one-applicable`, can also be specified by policies for combining decision results from enclosed rules. Note that all combining algorithms evaluate the enclosed elements, e.g. policies, in the order they are listed in the enclosing element, e.g. a policy set. Moreover, the implementation of the language permits also defining an algorithm named `custom-algorithm` in order to specify a customised way of combining decisions. This feature can be used, e.g., for having the `permit` decision if two and only two policies are evaluated to `permit`.

After this overview of the FACPL language, now we present in more detail the policies' structure and the meaning of their internal elements.

The example code previously reported shows a policy set specifying two policies and an obligation. Besides these elements, a policy set can also specify a `target` determining the requests to which the policy set applies. The syntax, as well as the semantics, of a *policy* is similar to that of a policy set. In particular, a policy encloses a set of rules instead of policies, as shown in the following code reporting a privacy consent policy for patient summaries.

```
Policy PatientSummary < permit-overrides
    target:
        equal ( "medical doctor" , subject/role ) && &
        equal ( "TREATEMENT" , subject/purposeofuse ) && &
        equal ( "34133-9" , resource/resource-id )
    rules:
        Rule rule1 ( permit
            target:
                equal ( "Read" , action/action-id )
            condition:
                subset ( string-bag("PRD-003","PRD-005","PRD-010","PRD-016"),
                         subject/hl7.permission)
        )
    obl:
        [ permit M log(subject/doctor-id, resource/resource-id)]
        [ deny   M log(resource/resource-id.email, "Your medical record
                       has been requested by EpSOS")]
>
```

As for policy sets, policy applicability is defined through the boolean expression in the `target`. This expression consists of applications of *matching functions*, i.e., functions comparing literal values and attribute values, composed through conjunctive and disjunctive boolean operators. For example, the function `equal` in the target checks the equality between the string `medical doctor` and the value, retrieved from the request, of the attribute `subject/role`. More precisely, the structure of a matching function application is

$$matchingFunction\ (\ literal\ ,\ structured\text{-}name\ )$$

where the two arguments have the following meaning:

- *literal* is the value (of type integer, double, boolean, string or date) to be compared with the attribute value from the request;
- *structured-name* identifies a request's attribute, e.g. `subject/role` indicates the attribute with identifier `role` in the category `subject`.

The following matching functions are available:

- `equal`
- `not-equal`
- `greater-than`
- `less-than`
- `greater-than-or-equal`
- `less-than-or-equal`

The evaluation of the target expression determines if the policy applies to a request and produces one of the values `match`, `no-match`, and `indeterminate`. The first two values have an obvious meaning, the last one is returned in case of errors. If there are conjunction or disjunction of matching functions' applications, the resulting value is obtained according to the truth tables reported in Table 1. Notably, when the target is omitted the enclosing element is applicable to all request.

The *obligations* permit defining additional actions to be performed for enforcing authorisation decisions. An obligation specifies an *effect*, i.e. `permit` or `deny`, for the applicability of the obligation, a *type*, i.e. `M` for *Mandatory* and `O` for *Optional*, and the action, with its arguments' expressions. The expressions have the same syntax as rules' conditions (which will be described shortly). These expressions must be successfully evaluated to literal values by the PDP for the corresponding obligations to be fulfilled. For example, the obligation below, drawn from the policy previously shown, requires to register in a log the event corresponding to the access to the resource by the doctor.

```
[permit M log(subject/doctor-id , resource/resource-id)]
```

Table 1. Target operators

&&	match	no-match	indeterminate
match	match	no-match	indeterminate
no-match	no-match	no-match	no-match
indeterminate	indeterminate	no-match	indeterminate

||	match	no-match	indeterminate
match	match	match	match
no-match	match	no-match	indeterminate
indeterminate	match	indeterminate	indeterminate

When this obligation is processed by the PDP the two expressions argument of the `log` action are evaluated, i.e. the structured names `subject/doctor-id` and `resource/resource-id` are used to retrieve from the request the values to be written in the log. Thus, in the obligation returned by the PDP, the two structured names will be replaced by the corresponding request's values.

The *rules* are the basic elements of the language. A rule specifies an *effect*, which indicates the rule-writer's intended consequence of a positive evaluation of the rule (the allowed values are `permit` and `deny`), a target, a condition (which is a boolean expression that may further refine the applicability of the rule), and a set of obligations. Target, condition and obligations may be missing, as in the rule

```
Rule denyAll(deny)
```

used in the policy set shown at page 4. A *condition* is a boolean expression formed by functions applied to values and attributes (the complete list of functions is reported in Table 2). It is more general than a target expression, which just consists of conjunctions and disjunctions of matching functions' applications. In fact, to define condition expressions, FACPL provides the main logical, relational, and arithmetic operators, which operate on boolean, string, integer, double, and date values. Moreover, bags (i.e., unordered collections that may contain duplicated values) of strings are also supported. For example, the rule of the privacy consent policy for checking the action `Read` and the requestor's permissions is as follows.

```
Rule rule1 ( permit
    target:
        equal ( "Read" , action/action-id )
    condition:
        subset ( string-bag("PRD-003","PRD-005","PRD-010","PRD-016"),
                 subject/hl7.permission)
    obl: )
```

The target checks that the action is a `Read` operation while the condition expression checks if the bag of values provided from the request comprises all permissions reported as literal, thus ensuring that the subject of the request has all permissions needed for reading the resource.

1.2 Evaluation Process

The process for calculating the authorisation decision for a given request starts from the PDP, which evaluates the request with respect to the stored policies. The evaluation of these policies is managed by the PDP's combining algorithm, which decides the strategy to follow for calculating the decision. Each algorithm executes a different workflow, e.g, `permit-overrides` (resp. `deny-overrides`) just stops its evaluation when an enclosed item, i.e. a policy set, a policy or a rule,

Table 2. Syntax of FACPL expressions

Expressions

Expression ::= *BoolExpr* | *StringExpr* | *ArithExpr* | *DateExpr*

Boolean expressions

BoolExpr ::= *Name* | `true` | `false`
| `and`(*BoolExpr*, ..., *BoolExpr*)
| `or`(*BoolExpr*, ..., *BoolExpr*)
| `not`(*BoolExpr*)
| `equal`(*Expression*, *Expression*)
| `greater-than`(*Expression*, *Expression*)
| `greater-than-or-equal`(*Expression*, *Expression*)
| `less-than`(*Expression*, *Expression*)
| `less-than-or-equal`(*Expression*, *Expression*)
| `at-least-one-member-of`(*BagExpr*, *BagExpr*)
| `subset`(*BagExpr*, *BagExpr*)

String expressions

StringExpr ::= *Name* | *StringValue*

BagExpr ::= `string-bag`(*StringExpr*, ..., *StringExpr*)

Arithmetic expressions

ArithExpr ::= *Name* | *IntegerValue* | *DoubleValue*
| `add`(*ArithExpr*, *ArithExpr*)
| `subtract`(*ArithExpr*, *ArithExpr*)
| `multiply`(*ArithExpr*, *ArithExpr*)
| `divide`(*ArithExpr*, *ArithExpr*)
| `mod`(*ArithExpr*, *ArithExpr*)
| `abs`(*ArithExpr*)

Date and time expressions

DateExpr ::= *Name* | *DateTimeValue*

Table 3. Conversion for extended indeterminate values

Combining Decision	Extended Indeterminate
`permit`	`indetP`
`deny`	`indetD`
`indeterminate`	`indetDP`
`indetP`	`indetP`
`indetD`	`indetD`
`indetDP`	`indetDP`
`not-applicable`	`not-applicable`

is evaluated to `permit` (resp. `deny`). We report the pseudo-code of the combining algorithms in Appendix B.

The evaluation of a policy starts from its target, which defines if the policy is applicable to the request. Thus, if the target evaluates to `match`, then the set of the enclosed rules are evaluated by following the policy's combining algorithm. Otherwise, if the target evaluates to `no-match`, the result is `not-applicable`. Finally, if the target evaluates to `indeterminate`, the evaluation of the policy returns the specific extended indeterminate value calculated by first evaluating the

enclosed rules and then applying the conversion rules reported in Table 3.For example, if the combining algorithm returns `permit` and the target evaluates to `indeterminate`, then the policy evaluates to `indetP`. This process can be summarised as in Table 4.

Table 4. Policy evaluation (without obligation)

Target	*Policy Value*
`match`	specified by the rule-combining algorithm
`no-match`	`not-applicable`
`indeterminate`	see Table 3

If the combined result includes obligations, these must be evaluated. Specifically, if the arguments of an obligation action cannot be successfully evaluated (e.g., the value of some attribute within an expression cannot be retrieved from the request or the context), the whole policy is evaluated to an indeterminate value, i.e. `indetP` if the combined result is `permit`, or `indetD` in case of `deny`. Otherwise, in case of success, the obligation is included in the decision statement and then passed through the levels of evaluation up to the top-level of the PDP, provided that the obligation's applicability effect is equal to the decision calculated at each evaluation level. In other words, if an obligation is defined in a rule with effect `permit` and an enclosing policy/policy set (not necessarily the tightest enclosing policy) is evaluated to `deny`, then this obligation is discarded and not passed to the upper level.

The evaluation of policy sets is the same as that of policies, but policies and policy sets are considered instead of rules.

The rule's evaluation process, shown in Table 5, starts with the target. If the target evaluation returns `match`, then the condition is evaluated. Now, if also the condition expression is successfully evaluated, i.e. its evaluation returns `true`, the evaluation of the rule returns the rule's *effect* as a result, i.e. either `permit` or `deny`. Instead, if the condition is `false` or the target is `no-match`, the decision is `not-applicable`. Finally, if any error occurs during the rule's evaluation, the result is an extended indeterminate value. Obligations possibly specified within the rule are evaluated similarly to those specified within the policies and policy sets.

Table 5. Rule evaluation (without obligation)

Target	*Condition*	*Rule Value*
`match`	`true`	*Effect*
`match`	`false`	`not-applicable`
`match`	`indeterminate`	`indetP` if *Effect* is `permit` `indetD` if *Effect* is `deny`
`no-match`	—	`not-applicable`
`indeterminate`	—	`indetP` if *Effect* is `permit` `indetD` if *Effect* is `deny`

It is worth noticing that the evaluation of an obligation can cause an `indeterminate` result even if the other components of the enclosing element are successfully evaluated. This may be due to, e.g, an attribute value that cannot be retrieved or an error in the evaluation of a function, such as a division by zero. These cases, for rules, policies and policy sets, are summarised in Table 6.

Table 6. Rule, policy and policy set evaluation with obligations

Rule/Policy/Policy Set Value	*Rule/Policy/Policy Set Final Value*
`permit`	`indetP` if an error occurs in obligations applicable in case of `permit` `permit` otherwise
`deny`	`indetD` if an error occurs in obligations applicable in case of `deny` `deny` otherwise
`not-applicable`	`not-applicable`
`indetDP`	`indetDP`
`indetP`	`indetP`
`indetD`	`indetD`

When the PDP completes its evaluation, the resulting decision is passed to the PEP, which is in charge of the enforcement process. For enforcing the decision, all obligations returned along with the PDP decision must be discharged, i.e. the PEP executes the specified actions. If such actions cannot be recognised by the PEP or other errors occur during their execution, the actions cannot be discharged. Notably, if an optional obligation, i.e. an obligation of type `O`, gives rise to an error, this error can be safely ignored. On the basis of the evaluation of all obligations, the PEP's enforcement algorithm applies the final authorisation decision.

2 The FACPL library

The FACPL code can be executed through a Java library that implements all the semantic tasks of the decision and the enforcement process briefly described in Section 1.2. In order to achieve a flexible and extendible framework, the library has been designed by exploiting the *reflection* features provided by Java and best-practice software engineering techniques. In fact, the framework can be easily extended to incorporate custom matching functions and combining algorithms defined by the user to deal with, e.g., new value types or specific decisions' combinations different from those described in Section 1. We first overview the library design and then show how a FACPL policy is transformed into a Java class in order to be involved in the PDP's evaluation process.

2.1 Library Design

The FACPL library is composed of three different components: one for the specification of rules, policies and policy sets, one for the specification of access requests and decision responses, and one for the implementations of the matching functions, of the combining algorithms, and of the decision and enforcing processes. For each of them, there are specific Java interfaces defining the signatures of the methods that must be implemented.

Rules, policies and policy sets are the elements evaluated by the PDP during the decision process. To specify these elements we have designed a class hierarchy, graphically depicted in Figure 1, implementing the interface `EvaluableElement`, which defines methods for calculating target matching and authorisation decision. All the evaluable elements share some common features, i.e. an identifier, a target, and a set of obligation expressions. Therefore, these features are factorized into the abstract class `PAFEvalElement`, which is the root of the class hierarchy.

The elements from which the decision process starts are policies and policy sets. Therefore, policies and policy sets are implemented via a common class `PAFElement`. The class for rules,

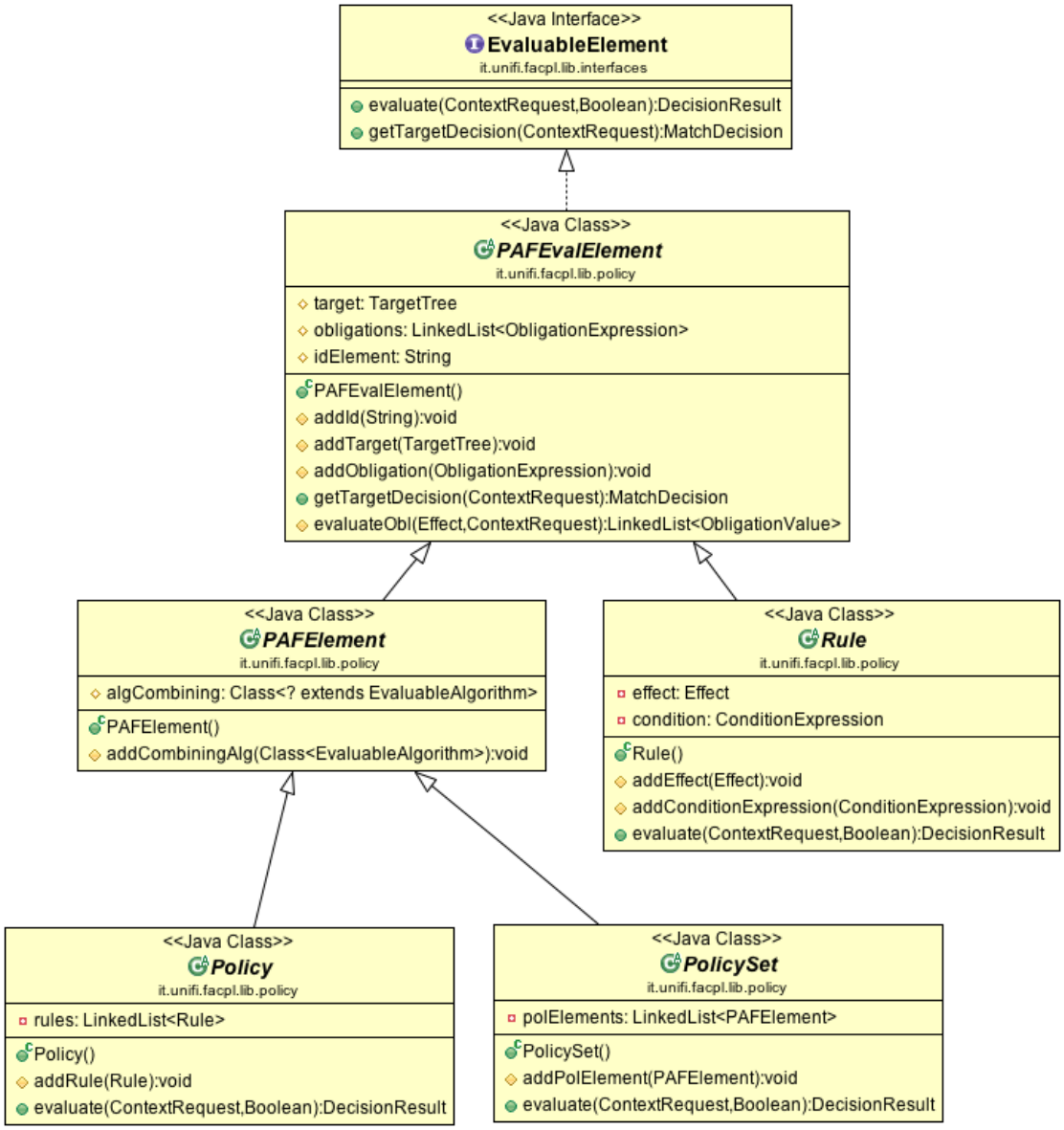

Fig. 1. Policy, policy set and rule hierarchy

instead, is a branch of the hierarchy. The class `PAFElement` contains the combining algorithm, while the class `Rule` contains the effect and the condition. The sub-classes `Policy` and `PolicySet` of `PAFElement` contain, respectively, the enclosed list of rules and of policies and/or policy sets. Each leaf class of the hierarchy, i.e. `Rule`, `Policy` and `PolicySet`, provides the method `evaluate` to calculate the element's `DecisionResult` value for the request passed as argument. The result contains the decision (i.e. `permit`, `deny`, `indetP`, etc.) and, possibly, a set of obligations.

For an easy and automatic creation of the Java classes from the FACPL code, the classes described above provide specific setting methods for each element's attribute, such as `addTarget` and `addCondition`. Notably, the language implementation permits using policy and policy set references, which are useful for modularising the code.

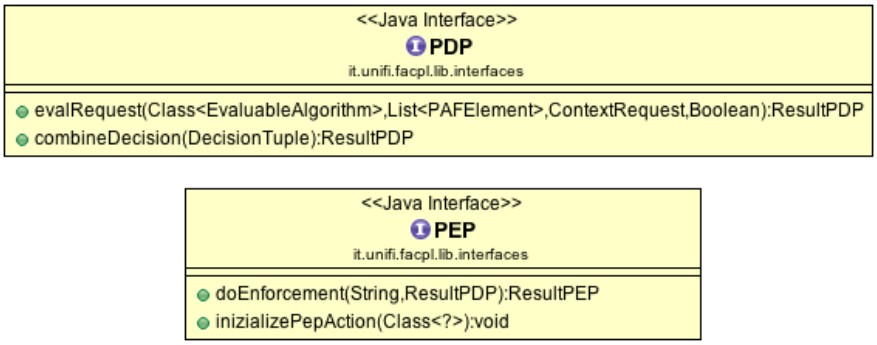

Fig. 2. PDP and PEP interfaces

The signature of PDP's methods is defined by the interface `PDP` shown in Figure 2. Specifically, it provides an entry method `evalRequest` for triggering the request evaluation, which takes as input a list of `PAFElement` objects[5], the PDP's combing algorithm and a request to evaluate. The method also requires a boolean argument indicating if the evaluation process has to use extended indeterminate values or not. Moreover, the interface defines the method `combineDecision` for combining multiple request decisions into one. The decision returned is, if exists, the common decision value, otherwise will be `indetDP`. Moreover, if there is any obligation attached to at least one decision of the set, the combined decision is `indetDP`. This is needed because an obligation is related to a specific decision, then in a common decision it could provide an erroneous behaviour. The FACPL library provides a standard implementation of PDP interface, faithful with the evaluation process presented in Section 1.2, through the class `PDP_Standard`.

The decision process is coordinated by the combining algorithm's implementation class passed as argument to the method `evalRequest` provided by `PDP`. The list of `PAFElements` and the access request to evaluate are then passed as arguments to the combining algorithm, which starts the decision process following its specific implementation. At the first call of the process, each element of the list is initialised and, hence, subsequent invocations of the decision process over the same policies do not require further initialisations. The outcome result of the PDP is an authorisation decision, which is passed to the PEP implementation class for enforcing it. Internally, the decision process supports the extended indeterminate values, but the final authorisation decision only contains the four authorisation values, i.e. the three indeterminate values are merged under the more general `indeterminate` decision. This avoids providing additional information about error causes to a malicious attacker that could exploit it for violating the access control system; the details about extended indeterminate decisions are however reported in a log file.

The functionalities supporting the enforcement process are defined by the `PEP` interface shown in Figure 2. This interface defines the method `doEnforcement` to enforce a PDP decision by discharging the additional actions possibly associated to the decision. The interface also specifies the method `initializePepAction` for loading enforcement actions from external classes at runtime. This permits adding new enforcement actions to the PEP definition without affecting the code of the class implementing it. The FACPL library provides a standard implementation of this interface, namely class `PEP_Standard`, which follows the workflow presented in Section 1.2. Specifically, it supports the algorithms `base`, `deny-biased` and `permit-biased`, and provides two basic enforcement actions: sending of emails and insertion of entries into log files. Anyway, as mentioned above, it is possible to define new enforcement actions, by exploiting the

[5] The class `PAFElement` has been indeed introduced in the class hierarchy in order to simplify the definition of the PDP implementation.

class `PEPAction`, and to add them to the current PEP implementation by means of method `initializePepAction`. In particular, for each additional action to add, we have to report in the `PEPAction` class the name and the reference to the class that implements it. This latter class is auto-generated by a Java code generator.

Access requests and responses of PDP and PEP are designed according to the syntax reported in Table 8. Moreover, each request is linked to its *context* through which it retrieves values from sources outside the request, i.e. we have an *explicit* context. In the eHealth scenario reported in Section 1, for example, the context is used to retrieve the redirecting URL requested by the evaluation of the obligation within the policy set. This feature can also be used to integrate our library with other services.

Both the decision and the enforcement processes are well-documented by log information, available in the standard output or in a dedicated file. The logging tool exploited by our library is the Simple Logging Facade for Java (SLF4J)[6] that permits to abstract from a single way of logging and supports multiple well-known logging libraries, e.g. Apache Log4j[7] or CommonsLogging[8].

The main advantage of our Java-based implementation, with respect to the usual XML-based ones, is to avoid parsing and surfing the XML tree structure of policies, which require an additional computational load.

2.2 An Example of Java Policy

From the description above, it is worth noticing that each FACPL element in the library corresponds to an abstract class, which provides a method for evaluating requests with respect to such element. Therefore, a FACPL policy is rendered as a Java class that extends the corresponding abstract class `Policy`. Policy elements, i.e. combining algorithm, target, rules and obligations, are added to a `Policy` object at creation time by the class constructor, by means of specific methods, e.g. `addTarget`. Policy sets are translated similarly. For example, an excerpt of the Java code corresponding to the *Patient Summary* policy is reported below.

```
public class Policy_PatientSummary extends Policy{

    public Policy_PatientSummary(){
        addId("PatientSummary");
        //Algorithm Combining
        addCombiningAlg(it.unifi.facpl.lib.algorithm.PermitOverrides.class);
        //Target
        addTarget(new TargetTree(Connector.AND,
        new TargetTree(
            new TargetExpression(it.unifi.facpl.lib.function.Equal.class, "TREATEMENT",new
                StructName("subject.access","purposeofuse"))),
         new TargetTree(Connector.AND,
            new TargetTree(new TargetExpression(it.unifi.facpl.lib.function.Equal.class, "
                medical doctor" ,new StructName("subject","role"))),
             new TargetTree(new TargetExpression(it.unifi.facpl.lib.function.Equal.class, "
                 34133-9",new StructName("resource","resource-id"))))
        ));
        //Rule
        addRule(new Rule1());
        //Obligations
        addObligation(new ObligationExpression("log",Effect.PERMIT,TypeObl.M,new StructName("
            subject","doctor-id"),new StructName("resource","resource-id")));
    }

    private class Rule1 extends Rule{

            Rule1 (){
```

[6] http://www.slf4j.org/
[7] http://logging.apache.org/log4j/1.2/
[8] http://commons.apache.org/proper/commons-logging/

```
                addId("rule1");
                //Effect
                addEffect(Effect.PERMIT);
                //Target
                addTarget(...);
                //Condition
                addConditionExpression(new ConditionExpression(it.unifi.facpl.lib.function.
                    StringSubset.class,
                new Bag("PRD-003","PRD-005","PRD-010","PRD-016")
                ,new StructName("subject","hl7.permission")));
        }
    }
}
```

The policy evaluation is coordinated by the class implementing the combing algorithm (i.e. `PermitOverrides.class`). The expression corresponding to the policy target is structured as nested expressions organised according to the structure of the original FACPL target (possibly defined by brackets). Since rules are only used inside their enclosing policy, for each of them the policy class contains an inner class (in the code above, the enclosed rule `rule1` is implemented by the inner class `Rule1`). As for policies, both rule and policy set classes have to extend the corresponding abstract class from the library.

For each FACPL request, instead, it is generated a class containing a list of attributes and a reference to a stub class corresponding to the context handler. By implementing this stub, it is possible to retrieve external information needed for evaluating requests. This can be exploited in those scenarios where the policy decision depends on the usage of system resources, e.g. condition expressions are used to check the amount of available resources in the system.

When all policies and requests have been translated into Java classes, it is possible to start the evaluation process by invoking the method `evalRequest` of the `PDP_Standard` class for calculating the authorization decision, and then the method `doEnfocement` of `PEP_Standard` for enforcing this decision. This workflow is provided by the FACPL IDE through the `main` method of the generated `PEP` class.

3 The FACPL IDE

The FACPL Integrated Development Environment (IDE) allows the policy designer to specify the system policies in FACPL. In addition to policies, the IDE permits also specifying user requests in order to test and validate the policies. The specification task is facilitated both by the high abstraction level of FACPL and by the graphical interface provided by our IDE. The toolchain supporting the use of FACPL is shown in Figure 3.

Specifically, the FACPL plugin provides both graphical and programming features, the most relevant are the following ones:

- code highlighting;
- code completion (called by *Ctrl+Space*);
- code navigation (outline and hyper-references across files);
- syntax and semantic checks (e.g., uniqueness of names, type checking for expression arguments);
- generation of Java code for requests' evaluation;
- generation of XACML policies (in XML style).

The IDE generates the corresponding low-level policies both in Java and in XML code by exploiting some translation rules, written on the top of the Xtext framework [6]. Such framework provides, indeed, facilities to develop environments for (domain specific) languages. The XML

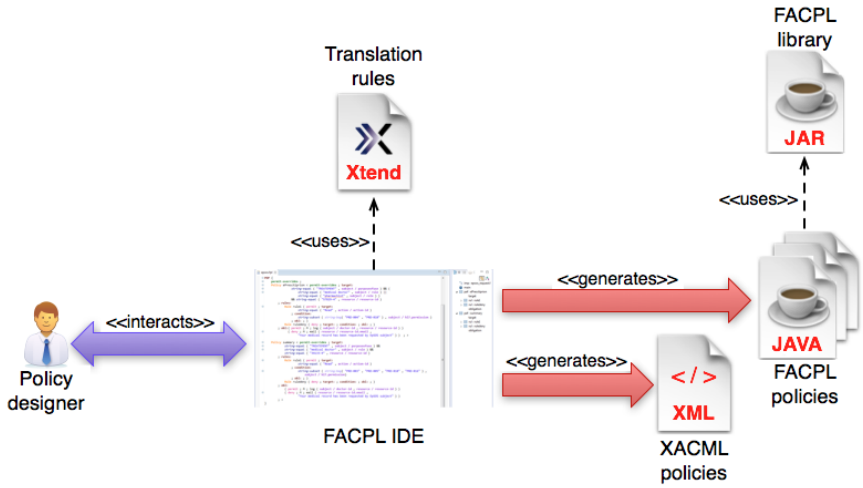

Fig. 3. FACPL toolchain

format obeys the XACML 3.0 syntax and can be used to connect our toolchain to external XACML tools. Instead, the other format relies on a Java library specifically designed for compile- and run-time supporting FACPL code.

In this section we firstly present the Xtext framework and then how to create, develop and test FACPL code inside the dedicated IDE environment, following a step-by-step presentation from plugin's installation to generation of Java code.

3.1 The Xtext Framework

The implementation of FACPL is developed with the Xtext [6] framework as an Eclipse plugin. Xtext provides a developing framework to design and implement domain specific languages. Starting from an EBNF formalization of the language syntax, such as the one for FACPL reported in [1], we can obtain an attribute grammar that defines the structure of the *abstract syntax tree* (AST) of the language. The AST is created by the ANTLR [4] parser and is merged into an Eclipse plugin by a specific workflow called MWE. Then the tree is specified as an Ecore Model [3], which is translated into a set of Java classes. Using these classes we can define syntax and semantic checks and a code generator for the language. Specifically the generator can be easily implemented with the Xtend language [5], which defines a specific generation rule for each syntactic category of the FACPL grammar. All these features are available as a plugin which provides a multi-page editor for the language and the generation features developed with Xtend. The plugin is created as an Eclipse RCP project, allowing additional customisations as any other plugin development project. However, the parser and the generator provided by Xtext and Xtend can be also called from other applications external to the deployed plugin.

3.2 Plugin Installation

The plugin is available on the FACPL website `http://rap.dsi.unifi.it/facpl` for Eclipse 4.* or higher equipped with the Xtext framework. These features, if absent, will be automatically added by the platform through the basic update site, e.g. for Juno is `http://download.eclipse.org/releases/juno`.

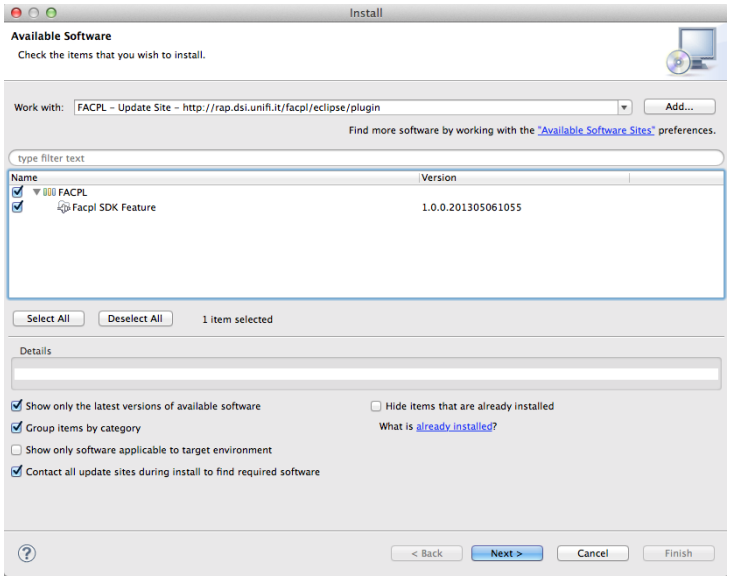

Fig. 4. Wizard for installing the FACPL plugin

To install the plugin we have to use the command "*Install new software ...*" inside the Eclipse's toolbar menu, in the outcoming panel we have to add the update site `http://rap.dsi.unifi.it/facpl/eclipse/plugin`. The corresponding panel is shown in Figure 4. In this panel all features of FACPL must be selected; then, after a few steps where it is simply required to accept the default options and after the platform restart, Eclipse completes the plugin installation.

3.3 Creation of a FACPL Project and a FACPL File

To create a new FACPL project, the FACPL IDE provides the customised wizard *new FACPL Project* from the menu *File -> New Project...*. A screenshot of this wizard is shown in Figure 5.

After choosing the project name, which cannot contain any blank space, we can click the finish button. The generated *FACPL Project* is a Java project where we have added the Xtext project characteristics, needed for using FACPL plugin features, and the libraries used for compiling and evaluating policies. A glimpse of an empty FACPL project is in Figure 6. Notably, the project includes a *config.properties* file, where parameters to execute enforcing actions are specified.

A new FACPL file can be created either as a new generic file with extension *".fpl"* or by means of the *new FACPL File* wizard from the menu *File − > New ...*. A screenshot of the wizard is shown in Figure 7. The wizard permits specifying the container of the file (by selecting it from the projects available in the workspace), the name of the file, and which templates of code to add. The three templates present in the wizard corresponds to the two parts of the FACPL language described in Section 1, i.e. PDP and request, and another additional component that contains some parameters for the project setting. Specifically, the meaning of the three wizard's options is as follows:

- **PDP**: it creates the code for the combining algorithm and the set of access control policies;
- **Main Attributes**: it adds a set of parameters for customising the generation of Java code and the authorisation/enforcement process;

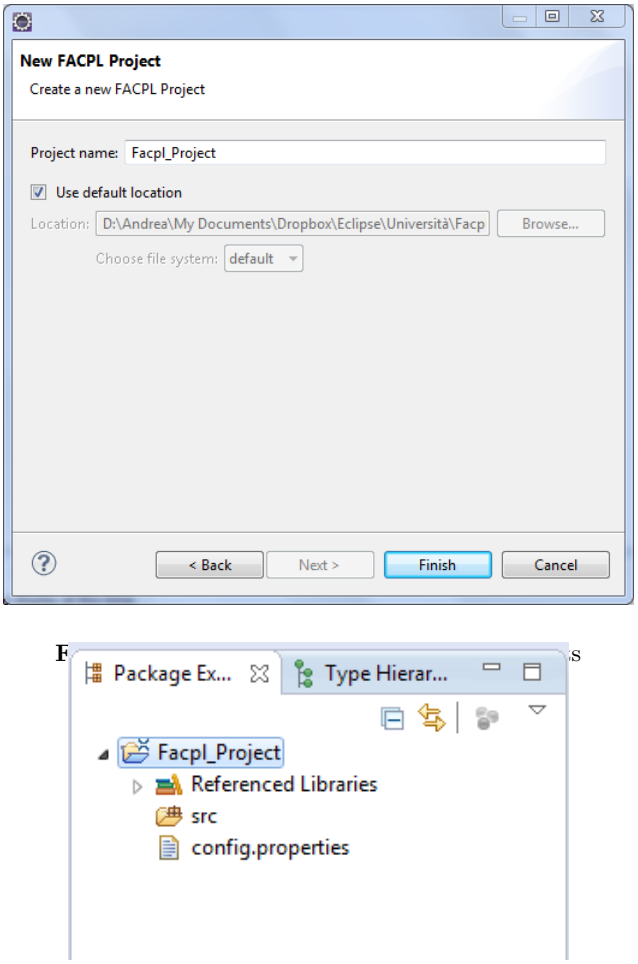

Fig. 6. Structure of an empty FACPL Project

– **Request**: it creates the code for specifying a set of access requests.

The FACPL file created by selecting all above items is shown in Figure 8, which in particular reports a view of the policy as shown by the IDE. The request and PAF parts of the code follow the specification of the FACPL language described in Section 1, while the meaning of Main attributes (the items at the top of Figure 8) is presented in the next section.

3.4 FACPL commands and facets

The FACPL plugin offers many facets to support policy development, from the organisation of code to the command for generating Java code. We present now how to use these useful features.

Navigation and formatting. The multi-page editor highlights FACPL keywords and policies' structure providing to each kind of item (i.e., combining algorithms, keywords, effects, and literals) a specific formatted syntax. Also the structure of policies is a key issue for understanding and maintaining the FACPL code. Thus, we allow a better code navigation by means of auto-indentation formatting. The standard Eclipse's command *Ctrl+Shift+F* can be used to call the

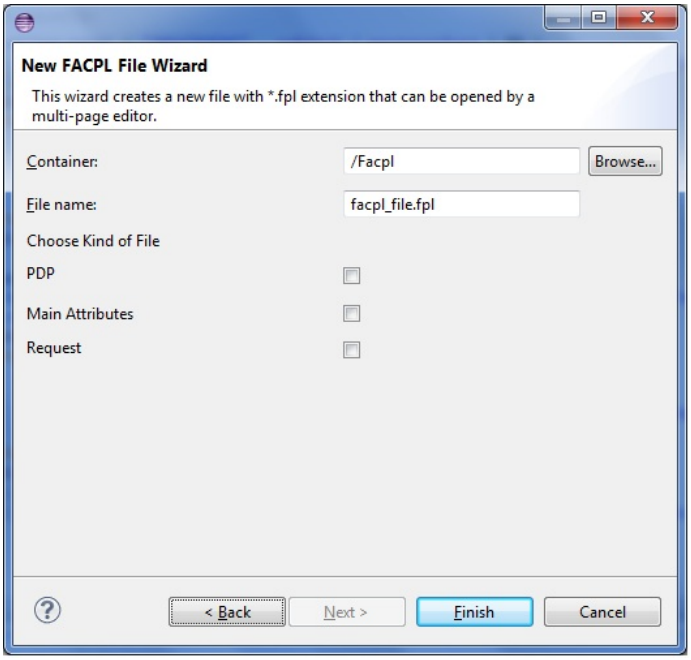

Fig. 7. Wizard for FACPL File

```
Combined Decision = false ;
Extended Indeterminate = false;
Generated Package = "" ;
Eval Request {NameRequest}


{
 pep: base
 pdp: deny-overrides
      PolicySet NamePolicySet {deny-overrides
          policies:
              Policy NamePolicy < deny-overrides
                   target: equal("value",category/id)
                        rules:
                        Rule ruleName (deny)
               obl: >
              obl:
      }
}

Request:{NameRequest
(category_attribute/id_attribute, "")}
```

Fig. 8. Template of a FACPL Policy

auto-formatter. The Structure of policies can be also navigated by means of the *Outline View* specifically designed for FACPL. An example of file's outline is in Figure 9.

Scopes and Import. The references to request in *Eval Request* set and to policy and policy set in *include* command rely on the current file's scope. The scope of a file is the set of requests,

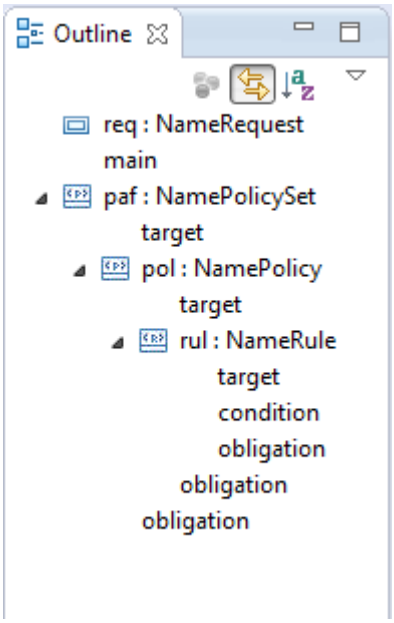

Fig. 9. FACPL project outline

policies and policy sets defined inside this file. In presence of *import* commands, file's scope is extended with the scope of the imported files. Namely, all requests, policies and policy sets defined in the imported file are also visible in the current file.

Name checks. For policies and policy sets it is defined a check for ensuring uniqueness of names. This check is performed among policy items together with policy set ones, because both of them can be used in an *include* command. When an *import* command is present, the name check verifies uniqueness of local items with respect to the imported ones.

Main attributes. The meaning of the attributes defined in the *main* section of the FACPL code (located at the top of the file) is as follows:

- *Combined Decision*: it permits requiring only one combined decision for a set of requests to be evaluated;
- *Extended Indeterminate*: it permits specifying whether the extended indeterminate values (i.e., `indetP`, `indetD`, `indetDP`) have to be used or not in the evaluation process (in the negative case, it is used the generic value `indeterminate`);
- *Generated Package*: it permits specifying the Java package enclosing the generated policy and request classes (if empty, it is assigned the default Java package);
- *Eval Request*: it reports the names of the set of requests to be evaluated (each request name must be identified into the file scope).

All these attributes must be defined together, i.e. it is not possible defining just one of them. When all attributes are correctly determinated, the Java code generator provides, in the *PEP* Java class, the *main* method for running requests' evaluation. Otherwise the generator creates all classes for requests, policies, etc., without providing the entry point method for evaluation.

Generation of Evaluable Code. For generating the evaluable Java code corresponding to a FACPL file, the FACPL IDE provides the command *Generate FACPL Code* from the pop-up

menu (right click in the editor or on the specific file in the package explorer) or from the *Facpl* toolbar menu. The resulting Java classes will be included in the package defined in the *main* attributes. If the current file does not specify any *main* attribute, no *main* method is generated in the PEP class. If there are one or more imported files, the generation command is also executed recursively on these files.

Generation of XACML (XML) policies. From the FACPL code it is possible to generate also the corresponding XACML files written as XML code. This facet is reachable by the command *Generate XML Code* from the pop-up menu or in the *Facpl* toolbar menu. The generated XML files are placed in the folder *src-xml* in the root of the project.

4 The eHealth Scenario

In this section we report the full policy set of the eHealth scenario, detailing each enclosed policies. Starting from this we show how these policies can be used for requests' evaluation, either with IDE and with Java. In the Listing 1.1 we report the FACPL code of all eHealth policies.

Listing 1.1. FACPL code for the eHealth scenario

```
{
    pep: base
    pdp: permit-overrides
     PolicySet eHealth {permit-overrides
        policies:
            Policy PatientSummary < permit-overrides  target:
                equal ( "TREATEMENT" , subject / purposeofuse ) & &
                equal ( "medical doctor" , subject / role ) & &
                equal ( "34133-9" , resource / resource-id )
            rules:
                Rule rule1 ( permit  target:
                    equal ( "Read" , action / action-id )
                 condition:
                    subset ( string-bag( "PRD-003" , "PRD-005" , "PRD-010" , "PRD-016" ),
                        subject / hl7.permission)
                 obl:  )
            obl:
                [ permit  M  log ( subject / doctor-id , resource / resource-id ) ]
            >
            Policy denyAll < permit-overrides
                rules:
                    Rule ruleDeny ( deny )
            >
        obl:
            [ permit M redirectTo(epsos/redirectto-url)]
            [ deny  M  mail ( resource / resource-id.email ,
            "Your medical record has been requested by EpSOS" ) ]
     }
}
```

The policy set follows the structure presented in Section 1. The request presented in Listing 1.2 depicts an access request granted to execute its action over the chosen resource.

Listing 1.2. FACPL code for a **granted** access request

```
Request:{Request1
    (subject/purposeofuse, "TREATEMENT")
    (subject/role, "medical doctor")
    (subject/doctor-id, "jh1234")
    (subject/hl7.permission, "PRD-003", "PRD-005", "PRD-010", "PRD-016")
    (resource/resource-id, "34133-9")
    (resource/resource-id.email, "mail@gmail.com")
    (action/action-id, "Read")
}
```

Analysing the previous request we can find all expected attributes' value required by the policies. Namely, there are the subject credentials needed for matching the target of `privacy consent` policy, the action `Read` required in the enclosed rule's target and each permission expected by the rule's condition expression. These assertions can be proof using the features of the FACPL IDE presented in Section 3, which we briefly summarise below.

Inside the IDE we create a FACPL project and then we write code of policy set and of request. The policy set and the request of Listing 1.1 and 1.2 can be written inside the IDE as they appear in listings. Calling the command *Generate Facpl Code* from toolbar menu of the IDE we generate the corresponding Java code, faithful to the class structure reported in Section 2.2. However, the Java code created cannot be evaluated because there is no main method in any classes. It is due to the absent of the main parameters in the policy code, in fact the generator will create only the class for specifying policies, request, and structure of PDP and PEP. Thus, we have to

collect each request and then call the PDP and pass its decision to the PEP. This task can be demanded to the code generator by adding the main parameters reported in Listing 1.3 inside the FACPL code.

Listing 1.3. FACPL parameters for creating **main** method

```
Combined Decision = false ;
Extended Indeterminate = true;
Generated Package = "eHealth";
Eval Request { Request1 }
```

With these parameters we set the destination package "*eHealth*" for Java code, the use of the extended indeterminate values and then the creation of the main method for evaluating the `Request1`. Obviously, if the `Request1` is defined in a separate file is needed an `import` command to reach it from the current file's scope. Running the generator the main method is added to the *PEP* class. For example to evaluate the `Request1` it is provided the code reported in Listing 1.4.

Listing 1.4. Java main commands for FACPL evaluation

```
public static void main(String[] args){
    //initialize PEP
    PEP.inizializePep(PEPAction.class);
    //log
    StringBuffer result = new StringBuffer();
    //request
    LinkedList<ContextRequest> requestes = new LinkedList<ContextRequest>();
    requestes.add(ContextRequest_Request1.getContextReq());
    for (ContextRequest rcxt : requestes) {
        ResultPDP resPDP = PDP.evalRequest(rcxt);
        result.append("Request: "+ resPDP.getId() + "\n");
        result.append("PDP Decision= " + resPDP.print()+"\n");
        //enforce decision
        ResultPEP resPEP = PEP.enforceDecision(resPDP);
        result.append("PEP Decision=" + resPEP.toString()+"\n");
    }
    System.out.println(result);
}
```

Firstly we initialise the available enforcement actions through the `PEPAction` class, secondly, for each request to evaluate, it is called the PDP's evaluation and then the resulting authorization decision is enforced through the PEP class. In the standard output is shown, as reported below, the results of the process.

```
Request: Request1

PDP Decision =  PERMIT   Obligation: Permit M log([jh1234, 34133-9])

PEP Decision =  PERMIT
```

Exploiting the classes `ResultPDP` and `ResultPEP` is possible to retrieve both level's decisions for applying them in the access control system.

A The grammar of FACPL

Table 7. Syntax of FACPL policies

Policy Authorization Framework

PAF ::= { `pdp` : PDP `pep` : PEP }

Policy Decision Point

PDP ::= {$Palg$ `policies` : $Policy^+$}

Policy-combining algorithms

$Palg$ ::= `only-one-applicable` | $Ralg$

Rule-combining algorithms

$Ralg$ ::= `deny-overrides` | `permit-overrides` | `first-applicable`
| `deny-unless-permit` | `permit-unless-deny` | `custom-algorithm`

Policies: atomic policies and policy sets

$Policy$ ::= `Policy` $Identifier$ ⟨$Ralg$ `target` : $Target^?$ `rules` : $Rule^+$ `obl` : $Obligation^*$ ⟩
| `PolicySet` $Identifier$ {$Palg$ `target` : $Target^?$ `policies` : $Policy^+$ `obl` : $Obligation^*$ }
| `include` $Identifier$

Rules

$Rule$ ::= `Rule` $Identifier$ ($Effect$ `target` : $Target^?$ `condition` : $BoolExpr^?$ `obl` : $Obligation^*$)

Effects

$Effect$ ::= `permit` | `deny`

Targets

$Target$ ::= $MatchId(Value,Name)$ | $Target \wedge Target$ | $Target \vee Target$

Matching functions

$MatchId$ ::= `equal` | `not-equal` | `greater-than` | `less-than`
| `greater-than-or-equal` | `less-than-or-equal`

Names

$Name$ ::= $Identifier/Identifier$

Obligations

$Obligation$ ::= [$Effect$ $Type$ $PepAction(Expression^*)$]

Type: mandatory or optional

$Type$ ::= `M` | `O`

Policy Enforcement Point

PEP ::= `base` | `deny-biased` | `permit-biased`

Table 8. Syntax of FACPL requests and responses

Request sets

$RequestSet \ ::= \ \{\, Request^{+} \,\}$

Requests

$Request \ ::= \ \texttt{request} : Identifier \ (Name, Value)^{+}$

Authorization decisions (PDP responses)

$AD \ ::= \ (\texttt{permit} : RequestWithOblS$
$\texttt{deny} : RequestWithOblS$
$\texttt{not-applicable} : \{\, Identifier^{*} \,\}$
$\texttt{indetP} : \{\, Identifier^{*} \,\}$
$\texttt{indetD} : \{\, Identifier^{*} \,\}$
$\texttt{indetDP} : \{\, Identifier^{*} \,\})$

Sets of requests with evaluated obligations

$RequestWithOblS \ ::= \ \{\, \langle\, Identifier \quad \texttt{evObl} : EvalObligation^{*} \rangle^{*} \,\}$

Evaluated obligations

$EvalObligation \ ::= \ [\, Type \quad PepAction(Value^{*}) \,]$

Enforceable decisions (PEP responses)

$ED \ ::= \ (\texttt{permit} : \{\, Identifier^{*} \,\}$
$\texttt{deny} : \{\, Identifier^{*} \,\}$
$\texttt{not-applicable} : \{\, Identifier^{*} \,\}$
$\texttt{indetP} : \{\, Identifier^{*} \,\}$
$\texttt{indetD} : \{\, Identifier^{*} \,\}$
$\texttt{indetDP} : \{\, Identifier^{*} \,\})$

B Combining algorithms

We report in this appendix the definitions of the combining algorithms not described in the paper. Specifically, we present only one code for both rule- and policy-combining algorithms; therefore, from time to time, *Sequence* stands for a sequence of rules, policies or policy sets, and `elem` for one of its elements. Notice that the instruction 'foreach (elem ∈ *Sequence*)' traverses the elements within *Sequence* in the order in which they are listed in the policy or policy set.

Listing 1.5. The algorithm `permit-overrides`

```
1  Boolean atLeastOneErrorD = false;
2  Boolean atLeastOneErrorP = false;
3  Boolean atLeastOneErrorDP = false;
4  Boolean atLeastOneDeny  = false;
5  foreach (elem ∈ Sequence) {
6     AuthDecision t = ⟦ elem ⟧{r};
7     if (r ∈ t↓deny) {
8        atLeastOneDeny = true;
9        continue;
10    }
11    if (r ∈ t↓permit) return permit;
12    if (r ∈ t↓not-applicable) continue;
13    if (r ∈ t↓indetD) {
14       atLeastOneErrorD = true;
```

```
15        continue;
16     }
17     if (r ∈ t↓indetP) {
18        atLeastOneErrorP = true;
19        continue;
20     }
21     if (r ∈ t↓indetDP) {
22        atLeastOneErrorDP = true;
23        continue;
24     }
25  }
26  if (atLeastOneErrorDP) return indetDP;
27  if (atLeastOneErrorP && (atLeastOneErrorD || atLeastOneDeny))
28     return indetDP;
29  if (atLeastOneErrorP) return indetP;
30  if (atLeastOneDeny) return deny;
31  if (atLeastOneErrorD) return indetD;
32  return not-applicable;
```

Listing 1.6. The combining algorithm `deny-overrides`

```
1  Boolean atLeastOneErrorD = false;
2  Boolean atLeastOneErrorP = false;
3  Boolean atLeastOneErrorDP = false;
4  Boolean atLeastOnePermit = false;
5  foreach (elem ∈ Sequence) {
6     AuthDecision t = ⟦ elem ⟧{r};
7     if (r ∈ t↓deny) return deny;
8     if (r ∈ t↓permit) {
9           atLeastOnePermit = true;
10          continue;
11    }
12    if (r ∈ t↓not-applicable) continue;
13    if (r ∈ t↓indetD) {
14       atLeastOneErrorD = true;
15       continue;
16    }
17    if (r ∈ t↓indetP) {
18       atLeastOneErrorP = true;
19       continue;
20    }
21    if (r ∈ t↓indetDP) {
22       atLeastOneErrorDP = true;
23       continue;
24    }
25 }
26 if (atLeastOneErrorDP) return indetDP;
27 if (atLeastOneErrorP && (atLeastOneErrorD || atLeastOnePermit)) return
       indetDP;
28 if (atLeastOneErrorD) return indetD;
29 if (atLeastOnePermit) return permit;
30 if (atLeastOneErrorP) return indetP;
31 return not-applicable;
```

Listing 1.7. Combining algorithm `deny-unless-permit`

```
foreach (elem ∈ Sequence) {
   AuthDecision t = ⟦ elem ⟧{r};
   if (r ∈ t↓permit) return permit;
}
return deny;
```

Listing 1.8. Combining algorithm `permit-unless-deny`

```
foreach (elem ∈ Sequence) {
   AuthDecision t = ⟦ elem ⟧{r};
   if (r ∈ t↓deny) return deny;
}
return permit;
```

Listing 1.9. Combining algorithm `first-applicable`

```
foreach (elem ∈ Sequence) {
   AuthDecision t = ⟦ elem ⟧{r};
   if (r ∈ t↓deny) return deny;
   if (r ∈ t↓permit) return permit;
   if (r ∈ t↓not-applicable) continue;
   if (r ∈ t↓indetP) return indetP;
   if (r ∈ t↓indetD) return indetD;
   if (r ∈ t↓indetDP) return indetDP;
}
return not-applicable;
```

Listing 1.10. Policy-Combining algorithm `only-one-applicable`

```
Boolean          atLeastOne  = false;
Policy           selectedPolicy = null;
MatchingTuple    matchTuple;
foreach (policy ∈ Policies) {
   matchTuple = ⟦ policy.target ⟧{r};
   if (r ∈ matchTuple↓indeterminate) return indeterminate;
   if (r ∈ matchTuple↓match) {
      if (atLeastOne) {
         return indeterminate;
      } else {
         atLeastOne = true;
         selectedPolicy = policy;
      }
   }
   if (r ∈ matchTuple↓no-match) continue;
}
if (atLeastOne) {
   return ⟦ selectedPolicy ⟧{r};
} else {
   return not-applicable;
}
```

References

1. Andrea Margheri, Massimiliano Masi, Rosario Pugliese, and Francesco Tiezzi. A Formal Software Engineering Approach to Policy-based Access Control. Technical report, DiSIA, Univ. Firenze, 2013. Available at `http://rap.dsi.unifi.it/facpl/research/Facpl-TR.pdf`.
2. OASIS XACML TC. eXtensible Access Control Markup Language (XACML) version 3.0 - Candidate OASIS Standard, September 2012.
3. The Eclipse Foundation open source community. Eclipse modeling framework project, 2009. `http://www.eclipse.org/modeling/emf/`.
4. Terence J. Parr and Russell W. Quong. ANTLR: A Predicated-LL(k) Parser Generator. *Software Practice and Experience*, 25:789–810, 1994.
5. Xtend2, 2012. `http://www.eclipse.org/xtend/`.
6. Xtext. *Language Development Made Easy.* `http://www.eclipse.org/Xtext/`.